Topic Number 2
Efficiency – Complexity - Algorithm Analysis

"bit twiddling: 1. (pejorative) An exercise in tuning (see *tune*) in which incredible amounts of time and effort go to produce little noticeable improvement, often with the result that the code becomes incomprehensible."

- The Hackers Dictionary, version 4.4.7

Clicker Question 1

- "A program finds all the prime numbers between 2 and 1,000,000,000 from scratch in 0.37 seconds."
  - Is this a fast solution?

A. no

B. yes

C. it depends

Efficiency

- Computer Scientists don't just write programs.
- They also ***analyze*** them.
- How efficient is a program?
  - How much time does it take program to complete?
  - How much memory does a program use?
  - How do these change as the amount of data changes?
  - What is the difference between the average case and worst case efficiency if any?

Technique

- Informal approach for this class
  - more formal techniques in theory classes, CS331
- **How many computations will this program (method, algorithm) perform to get the answer?**
- Many simplifications
  - view algorithms as Java programs
  - **determine by analysis the total number executable statements (computations) in program or method as a function of the amount of data**
  - focus on the *dominant term* in the function

$T(N) = 17.5N^3 + 25N^2 + 35N + 251$ ***IS ORDER*** N^3

Counting Statements

```
int x; // one statement
x = 12; // one statement
int y = z * x + 3 % 5 * x / i; // 1
x++; // one statement
boolean p = x < y && y % 2 == 0 ||
               z >= y * x; // 1
int[] data = new int[100]; // 100
data[50] = x * x + y * y; // 1
```


Clicker 2

- What is output by the following code?

```
int total = 0;
for (int i = 0; i < 13; i++)
    for (int j = 0; j < 11; j++)
        total += 2;
System.out.println(total);
```

A. 24
B. 120
C. 143
D. 286
E. 338

Clicker 3

- What is output when method `sample` is called?

```
// pre: n >= 0, m >= 0
public static void sample(int n, int m) {
    int total = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            total += 5;
    System.out.println(total);
}
```

A. 5
B. n * m
C. n * m * 5
D. n^m
E. $(n * m)^5$

Example

```
public int total(int[] values) {
   int result = 0;
   for (int i = 0; i < values.length; i++)
      result += values[i];
   return result;
}
```

- How many statements are executed by method `total` as a function of `values.length`
- Let N = `values.length`
  - N is commonly used as a variable that denotes the amount of data

Counting Up Statements

- `int result = 0;` 1
- `int i = 0;` 1
- `i < values.length;` N + 1
- `i++` N
- `result += values[i];` N
- `return total;` 1
- T(N) = 3N + 4
- T(N) is the number of executable statements in method `total` as function of `values.length`

Another Simplification

- When determining complexity of an algorithm we want to simplify things
  - ignore some details to make comparisons easier
- Like assigning your grade for course
  - At the end of CS314 your transcript won't list all the details of your performance in the course
  - it won't list scores on all assignments, quizzes, and tests
  - simply a letter grade, B- or A or D+
- So we focus on the dominant term from the function and ignore the coefficient

Big O

- The most common method and notation for discussing the execution time of algorithms is *Big O*, also spoken *Order*
- Big O is the *asymptotic execution time* of the algorithm
  - In other words, how does the running time of the algorithm grow as a function of the amount of input data?
- Big O is an upper bounds
- It is a mathematical tool
- Hide a lot of unimportant details by assigning a simple grade (function) to algorithms

Formal Definition of Big O

- T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm. (The typical Big functions table)
  - c and N_0 are constants

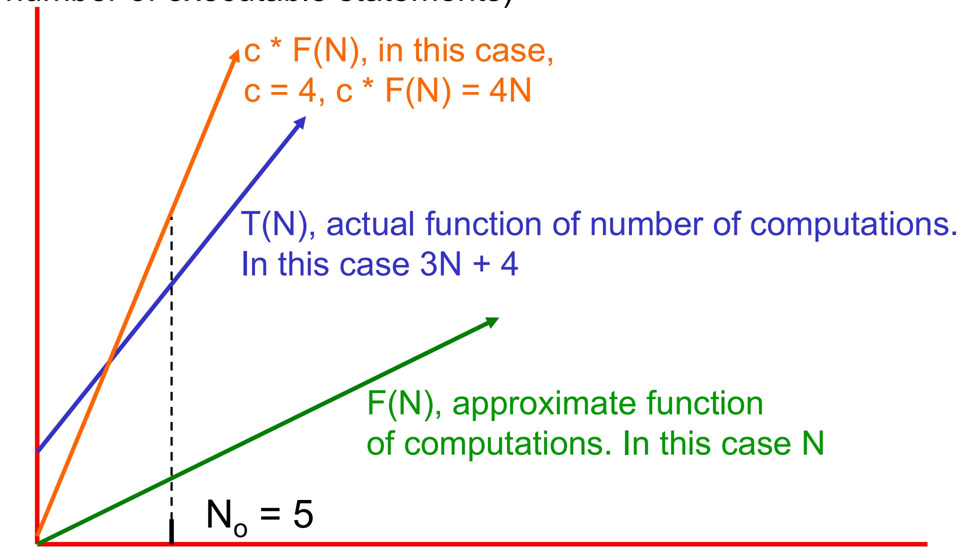
What it Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*

Showing O(N) is Correct

- Recall the formal definition of Big O
  - T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N > N_0$
- Recall method `total`, T(N) = 3N + 4
  - show method `total` is O(N).
  - F(N) is N
- We need to choose constants c and N_0
- how about c = 4, N_0 = 5 ?

vertical axis: time for algorithm to complete. (simplified to number of executable statements)



horizontal axis: N, number of elements in data set

Typical Big O Functions – "Grades"

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
N log N	N log N
N	Linear
$\sqrt{N}$	Root - n
log N	Logarithmic
1	Constant

Running time grows 'quickly' with more input.

Running time grows 'slowly' with more input.

Clicker 4

- Which of the following is true?
  Recall $T(N)_{total} = 3N + 4$

A. Method `total` is $O(N^{1/2})$

B. Method `total` is $O(N)$

C. Method `total` is $O(N^2)$

D. Two of A – C are correct

E. All of three of A – C are correct

Showing Order More Formally …

- Show $10N^2 + 15N$ is $O(N^2)$
- Break into terms.

- $10N^2 \le 10N^2$
- $15N \le 15N^2$ for $N \ge 1$ (Now add)
- $10N^2 + 15N \le 10N^2 + 15N^2$ for $N \ge 1$
- $10N^2 + 15N \le 25N^2$ for $N \ge 1$
- $c = 25$, $N_0 = 1$
- Note, the choices for c and N_0 are not unique.

Dealing with other methods

- What do I do about method calls?

```
double sum = 0.0;
for (int i = 0; i < n; i++)
    sum += Math.sqrt(i);
```

- Long way
  – go to that method or constructor and count statements
- Short way
  – substitute the simplified Big O function for that method.
  – **if** Math.sqrt is constant time, O(1), simply count `sum += Math.sqrt(i);` as one statement.

Dealing With Other Methods

```
public int foo(int[] data) {
    int total = 0;
    for (int i = 0; i < data.length; i++)
        total += countDups(data[i], data);
    return total;
}
// method countDups is O(N) where N is the
// length of the array it is passed
```

Clicker 5, What is the Big O of `foo`?

A. $O(1)$ B. $O(N)$ C. $O(N\log N)$

D. $O(N^2)$ E. $O(N!)$

Independent Loops

```
// from the Matrix class
public void scale(int factor) {
     for (int r = 0; r < numRows(); r++)
          for (int c = 0; c < numCols(); c++)
               iCells[r][c] *= factor;
}
```

`numRows()` returns number of rows in the matrix iCells

`numCols()` returns number of columns in the matrix iCells

Assume iCells is an N by N square matrix.
Assume numRows and numCols are O(1)

What is the T(N)? Clicker 6, What is the Order?

A. O(1) B. O(N) C. O(NlogN)
D. O(N^2) E. O(N!)

Bonus question. What if numRows is O(N)?

Just Count Loops, Right?

```
// Assume mat is a 2d array of booleans.
// Assume mat is square with N rows,
// and N columns.
public static void count(boolean[][] mat,
                              int row, int col) {
int numThings = 0;
for (int r = row - 1; r <= row + 1; r++)
     for (int c = col - 1; c <= col + 1; c++)
          if (mat[r][c])
               numThings++;
```

Clicker 7, What is the order of the method count?

A. O(1) B. O($N^{0.5}$) C. O(N) D. O(N^2) E. O(N^3)

It is **Not** Just Counting Loops

```
// "Unroll" the loop of method count:
int numThings = 0;
if (mat[r-1][c-1]) numThings++;
if (mat[r-1][c]) numThings++;
if (mat[r-1][c+1]) numThings++;
if (mat[r][c-1]) numThings++;
if (mat[r][c]) numThings++;
if (mat[r][c+1]) numThings++;
if (mat[r+1][c-1]) numThings++;
if (mat[r+1][c]) numThings++;
if (mat[r+1][c+1]) numThings++;
```


Just Count Loops, Right?

```
private static void mystery(int[] data) {
     stopIndex = data.length - 1;
     int j = 1;
     while (stopIndex > 0) {
          if (data[j - 1] > data[j]) {
               int t = data[j];
               data[j] = data[j - 1];
               data[j - 1] = t;
          }
          if (j == stopIndex) {
               stopIndex--;
               j = 1;
          } else {
               j++;
          }
     }
```

N = data.length

Clicker 8, What is the order of method mystery?

A. O(1) B. O($N^{0.5}$) C. O(N) D. O(N^2) E. O(N^3)

Sidetrack, the logarithm

- Thanks to Dr. Math
- $3^2 = 9$
- likewise $\log_3 9 = 2$
  - "The log to the base 3 of 9 is 2."
- The way to think about log is:
  - "the log to the base x of y is the number you can raise x to to get y."
  - Say to yourself "The log is the exponent." (and say it over and over until you believe it.)
  - In CS we work with base 2 logs, a lot
- $\log_2 32 = ?$ $\log_2 8 = ?$ $\log_2 1024 = ?$ $\log_{10} 1000 = ?$

When Do Logarithms Occur

- Algorithms tend to have a logarithmic term when they use a divide and conquer technique
- the size of the data set keeps getting divided by 2

```
public int foo(int n) {
    // pre n > 0
    int total = 0;
    while (n > 0) {
        n = n / 2;
        total++;
    }
    return total;
}
```

- Clicker 9, What is the order of the above code?

A. O(1) B. O(logN) C. O(N)
D. O(Nlog N) E. $O(N^2)$

The base of the log is typically not included as we can switch from one base to another by multiplying by a constant factor.

Significant Improvement – Algorithm with Smaller Big O function

- Problem: Given an array of ints replace any element equal to 0 with the maximum positive value to the right of that element. (if no positive value to the right, leave unchanged.)

Given:

[0, 9, 0, 13, 0, 0, 7, 1, -1, 0, 1, 0]

Becomes:

[**13**, 9, **13**, 13, **7**, **7**, 7, 1, -1, **1**, 1, 0]

Replace Zeros – Typical Solution

```
public void replace0s(int[] data){
  for(int i = 0; i < data.length; i++){
    if (data[i] == 0) {
      int max = 0;
      for(int j = i+1; j<data.length; j++)
         max = Math.max(max, data[j]);
      data[i] = max;
    }
  }
}
```

Assume all values are zeros. (worst case)
Example of a **dependent loops**.
Clicker 10 - Number of times j < data.length evaluated?

A.O(1) B. O(N) C. O(NlogN)
D. $O(N^2)$ E. O(N!)

Replace Zeros – Alternate Solution

```
public void replace0s(int[] data){
    int max =
       Math.max(0, data[data.length – 1]);
    int start = data.length – 2;
    for (int i = start; i >= 0; i--) {
        if (data[i] == 0)
            data[i] = max;
        else
            max = Math.max(max, data[i]);
  }
}
```

Clicker 11 - Big O of this approach?

A. O(1) B. O(N) C. O(NlogN)

D. O(N^2) E. O(N!)

Clicker 12

- Is O(N) really that much faster than O(N^2)?

A. never

B. always

C. typically

- Depends on the actual functions and the value of N.
- 1000N + 250 compared to N^2 + 10
- When do we use mechanized computation?
- N = 100,000
- 100,000,250 < 10,000,000,010 ($10^8 < 10^{10}$)

A VERY Useful Proportion

- Since F(N) is characterizes the running time of an algorithm the following proportion should hold true:

$F(N_0) / F(N_1) \sim= time_0 / time_1$

- An algorithm that is O(N^2) takes 3 seconds to run given 10,000 pieces of data.
  - How long do you expect it to take when there are 30,000 pieces of data?
  - common mistake
  - logarithms?

Why Use Big O?

- As we build data structures Big O is the tool we will use to decide under what conditions one data structure is better than another
- Think about performance when there is a lot of data.
  - "It worked so well with small data sets..."
  - Joel Spolsky, Schlemiel the painter's Algorithm
- Lots of trade offs
  - some data structures good for certain types of problems, bad for other types
  - often able to trade SPACE for TIME.
  - Faster solution that uses more space
  - Slower solution that uses less space

Big O Space

- Big O could be used to specify how much space is needed for a particular algorithm
  - in other words how many variables are needed
- Often there is a *time – space tradeoff*
  - can often take less time if willing to use more memory
  - can often use less memory if willing to take longer
  - truly beautiful solutions take less time and space

The biggest difference between time and space is that you can't reuse time. - Merrick Furst

Quantifiers on Big O

- It is often useful to discuss different cases for an algorithm
- Best Case: what is the best we can hope for?
  - least interesting, but a good exercise
  - **Don't assume no data. Amount of date is still variable, possibly quite large**
- Average Case (a.k.a. expected running time): what usually happens with the algorithm?
- Worst Case: what is the worst we can expect of the algorithm?
  - very interesting to compare this to the average case

Best, Average, Worst Case

- To Determine the best, average, and worst case Big O we must make assumptions about the data set
- Best case -> what are the properties of the data set that will lead to the fewest number of executable statements (steps in the algorithm)
- Worst case -> what are the properties of the data set that will lead to the largest number of executable statements
- Average case -> Usually this means assuming the data is randomly distributed
  - or if I ran the algorithm a large number of times with different sets of data what would the average amount of work be for those runs?

Another Example

```
public double minimum(double[] values) {
   int n = values.length;
   double minValue = values[0];
   for (int i = 1; i < n; i++)
      if (values[i] < minValue)
         minValue = values[i];
   return minValue;
}
```

- T(N)? F(N)? Big O? Best case? Worst Case? Average Case?
- If no other information, assume asking average case

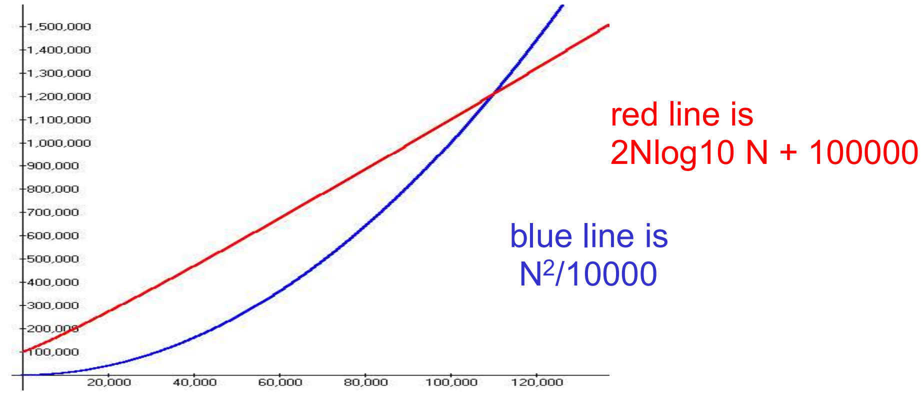
Example of Dominance

- Look at an extreme example. Assume the actual number as a function of the amount of data is:

  $N^2/10000 + 2N\log_{10} N + 100000$

- Is it plausible to say the N^2 term dominates even though it is divided by 10000 and that the algorithm is $O(N^2)$?
- What if we separate the equation into $(N^2/10000)$ and $(2N \log_{10} N + 100000)$ and graph the results.

Summing Execution Times



- For large values of N the N^2 term dominates so the algorithm is $O(N^2)$
- When does it make sense to use a computer?

Comparing Grades

- Assume we have a problem
- Algorithm A solves the problem correctly and is $O(N^2)$
- Algorithm B solves the same problem correctly and is $O(N \log_2 N)$
- Which algorithm is faster?
- One of the assumptions of Big O is that the data set is large.
- The "grades" should be accurate tools if this holds true.

Running Times

- Assume N = 100,000 and processor speed is 1,000,000,000 operations per second

Function	Running Time
2^N	$3.2 \times 10^{30,086}$ years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
$N\sqrt{N}$	0.032 seconds
N log N	0.0017 seconds
N	0.0001 seconds
$\sqrt{N}$	3.2×10^{-7} seconds
log N	1.2×10^{-8} seconds

Theory to Practice OR Dykstra says: "Pictures are for the Weak."

	1000	2000	4000	8000	16000	32000	64000	128K
O(N)	2.2×10^{-5}	2.7×10^{-5}	5.4×10^{-5}	4.2×10^{-5}	6.8×10^{-5}	1.2×10^{-4}	2.3×10^{-4}	5.1×10^{-4}
O(NlogN)	8.5×10^{-5}	1.9×10^{-4}	3.7×10^{-4}	4.7×10^{-4}	1.0×10^{-3}	2.1×10^{-3}	4.6×10^{-3}	1.2×10^{-2}
$O(N^{3/2})$	3.5×10^{-5}	6.9×10^{-4}	1.7×10^{-3}	5.0×10^{-3}	1.4×10^{-2}	3.8×10^{-2}	0.11	0.30
$O(N^2)$ ind.	3.4×10^{-3}	1.4×10^{-3}	4.4×10^{-3}	0.22	0.86	3.45	13.79	(55)
$O(N^2)$ dep.	1.8×10^{-3}	7.1×10^{-3}	2.7×10^{-2}	0.11	0.43	1.73	6.90	(27.6)
$O(N^3)$	3.40	27.26	(218)	(1745) 29 min.	(13,957) 233 min	(112k) 31 hrs	(896k) 10 days	(7.2m) 80 days

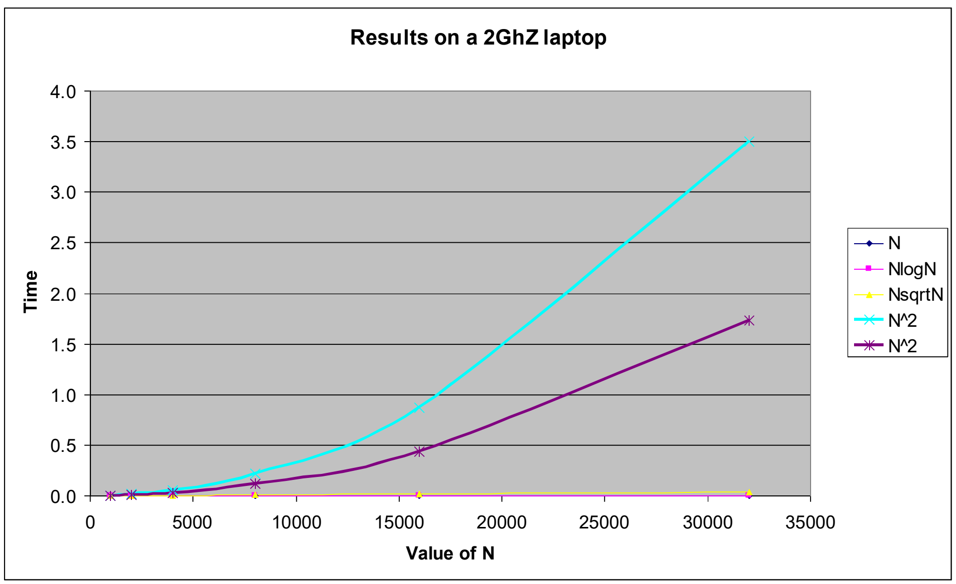
Times in Seconds. Red indicates predicated value.

Change between Data Points

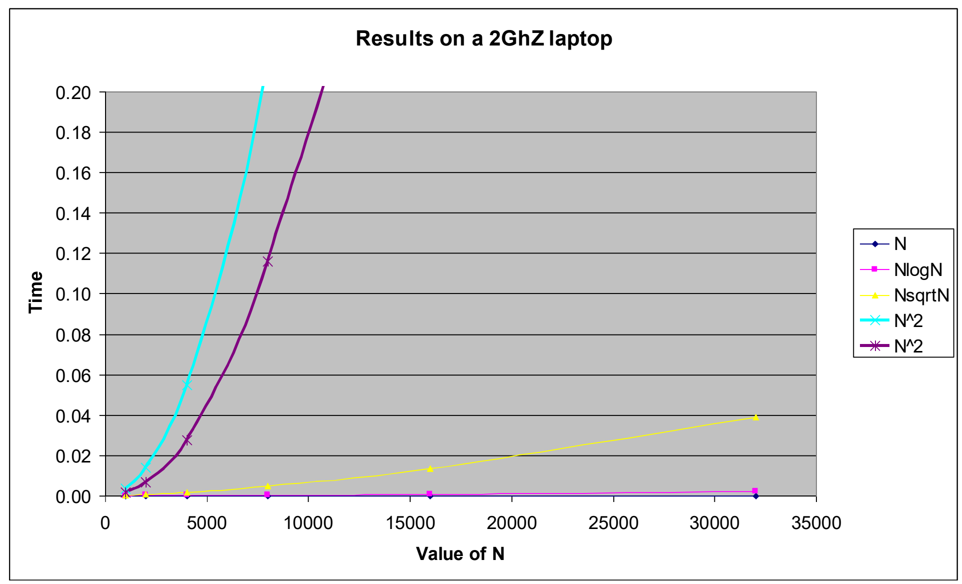
	1000	2000	4000	8000	16000	32000	64000	128K	256k	512k
O(N)	-	1.21	2.02	0.78	1.62	1.76	1.89	2.24	2.11	1.62
O(NlogN)	-	2.18	1.99	1.27	2.13	2.15	2.15	2.71	1.64	2.40
$O(N^{3/2})$	-	1.98	2.48	2.87	2.79	2.76	2.85	2.79	2.82	2.81
$O(N^2)$ ind	-	4.06	3.98	3.94	3.99	4.00	3.99	-	-	-
$O(N^2)$ dep	-	4.00	3.82	3.97	4.00	4.01	3.98	-	-	-
$O(N^3)$	-	8.03	-	-	-	-	-	-	-	-

Value obtained by $Time_x$ / $Time_{x-1}$

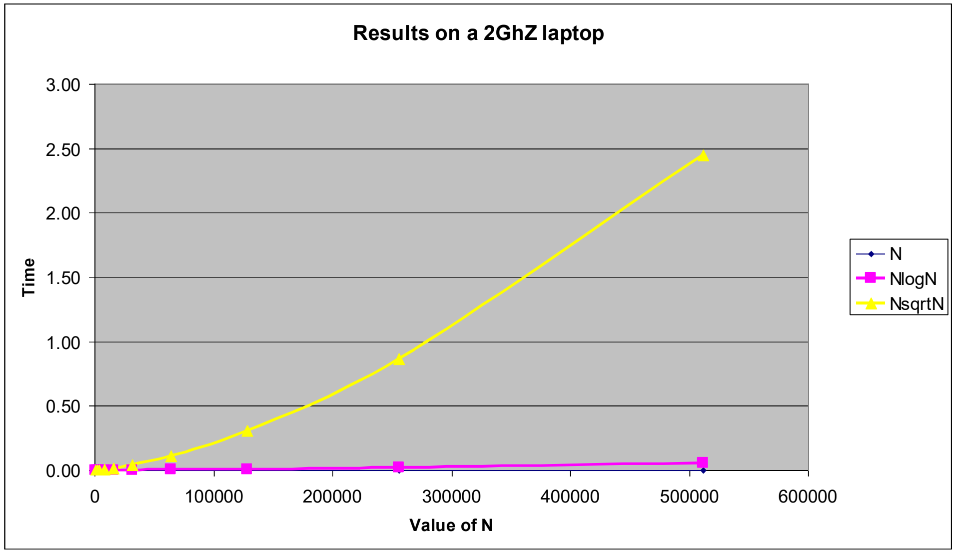
Okay, Pictures



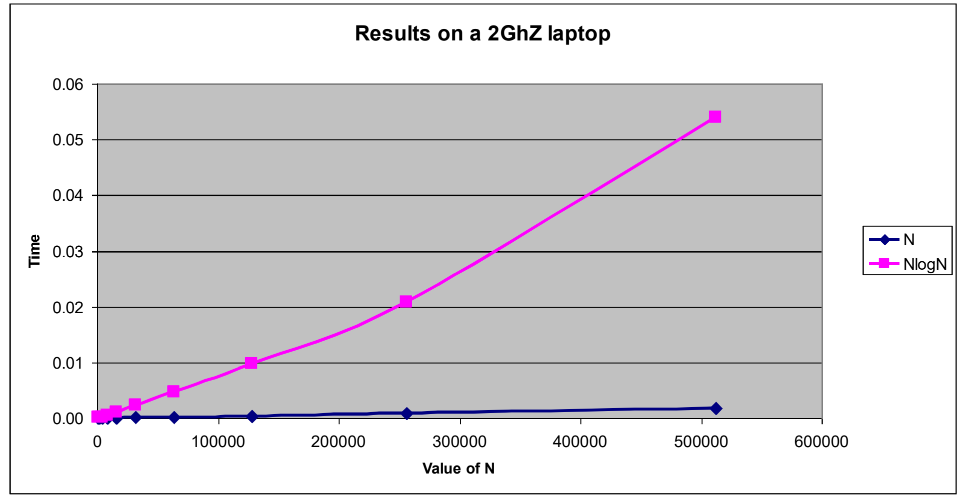
Put a Cap on Time



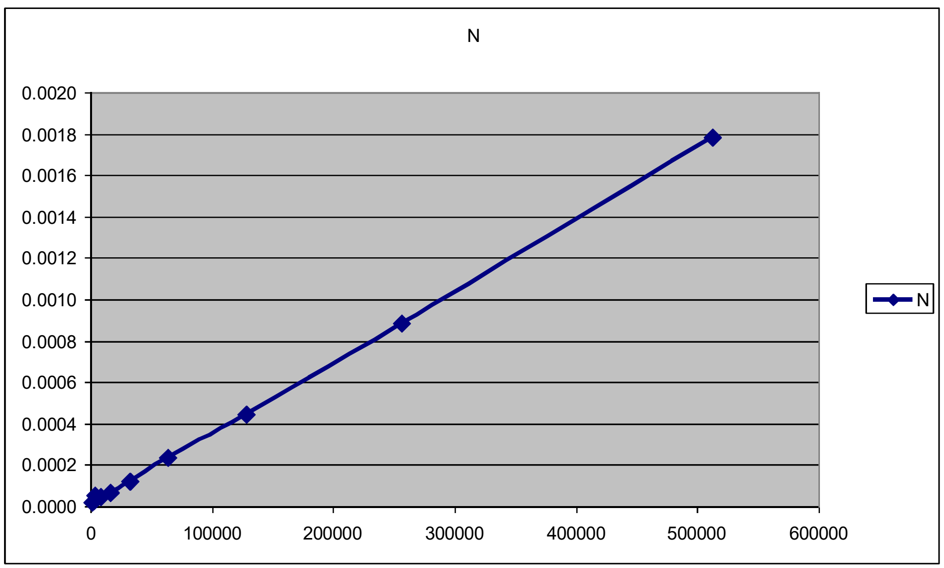
No O(N^2) Data



Just O(N) and O(NlogN)



Just O(N)



10^9 instructions/sec, runtimes

N	*O(log N)*	*O(N)*	*O(N log N)*	*O(N²)*
10	0.000000003	0.00000001	0.000000033	0.0000001
100	0.000000007	0.00000010	0.000000664	0.0001000
1,000	0.000000010	0.00000100	0.000010000	0.001
10,000	0.000000013	0.00001000	0.000132900	0.1 min
100,000	0.000000017	0.00010000	0.001661000	10 seconds
1,000,000	0.000000020	0.001	0.0199	16.7 minutes
1,000,000,000	0.000000030	1.0 second	30 seconds	31.7 years

Formal Definition of Big O (repeated)

- T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm
  - c and N_0 are constants

More on the Formal Definition

- There is a point N_0 such that for all values of N that are past this point, T(N) is bounded by some multiple of F(N)
- Thus if T(N) of the algorithm is O(N^2) then, ignoring constants, at some point we can *bound* the running time by a quadratic function.
- given a *linear* algorithm it is *technically correct* to say the running time is O(N ^ 2). O(N) is a more precise answer as to the Big O of the linear algorithm
  - thus the caveat "pick the most restrictive function" in Big O type questions.

What it All Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*

Other Algorithmic Analysis Tools

- *Big Omega* T(N) is Ω(F(N)) if there are positive constants c and N_0 such that $T(N) \geq cF(N)$ when $N \geq N_0$
  - Big O is similar to less than or equal, an upper bounds
  - Big Omega is similar to greater than or equal, a lower bound
- *Big Theta* T(N) is θ(F(N)) if and only if T(N) is O(F(N))and T(N) is Ω(F(N)).
  - Big Theta is similar to equals

Relative Rates of Growth

Analysis Type	Mathematical Expression	Relative Rates of Growth
Big O	$T(N) = O(F(N))$	$T(N) \leq F(N)$
Big Ω	$T(N) = \Omega(F(N))$	$T(N) \geq F(N)$
Big θ	$T(N) = \theta(F(N))$	$T(N) = F(N)$

"In spite of the additional precision offered by Big Theta, Big O is more commonly used, except by researchers in the algorithms analysis field" - Mark Weiss